Towards Differentiation-Enabled Fortran 95 Compiler Technology

Malcolm Cohen
The Numerical Algorithms Group, Oxford, UK
Malcom.Cohen@nag.co.uk

Uwe Naumann
Argonne National Laboratory, IL, USA
naumann@mcs.anl.gov

Jan Riehme
University of Hertfordshire, Hatfield, UK
riehme@math.tu-dresden.de

ABSTRACT

We present a novel approach to generating derivative code for mathematical models implemented as Fortran 95 programs using Automatic Differentiation inside a compiler. This technique allows us to combine the advantages of both operator overloading and source transformation based tools for Automatic Differentiation. Furthermore, the compiler's infrastructure for syntactic, semantic, and static data flow analysis can be built on.

Keywords

Compiler, Automatic Differentiation

1. INTRODUCTION

Today, the mathematical models of a large number of scientific, engineering, and socio-economic applications are given as computer programs written in some high-level programming language, such as Fortran 95. In order to make the highly desirable transition from the exclusive simulation of the underlying real-world processes to a systematic optimization of both the model and the corresponding application derivative information of the model outputs with respect to certain parameters is required. Straight-forward numerical differentiation using divided difference quotients is often just not good enough. It may either lack the required accuracy or the possibly very large number of parameters leads to an unrespectable computational complexity. A remedy for these difficulties is Automatic Differentiation (AD) [11, 7, 10]. Based on the well-known partial derivatives of the elementary arithmetic operators $(+, -, *, /)$ and intrinsic functions $(\sin, \cos, \exp, \ldots)$ with respect to their arguments the chain rule is used to compute first and possibly higher-order derivatives. This can be implemented as a source transformation process which creates a derivative code for a given program. Consider, for example, the following simple program implementing a vector function $F : \mathbb{R}^{12} \to \mathbb{R}^{10}$ where $(\mathtt{a}, \mathtt{x}, \mathtt{y(10)}) \mapsto \mathtt{y}$.

```
do i=1,10
  if (i>1) then
    h=x(i)*y(i-1)
  else
    h=a*x(i)+y(10)
  endif
  y(i)=sin(h)
end do
```

For brevity, the declaration and initialization part has been omitted. Think of the above as the body of a subroutine `SUB` with `REAL` arguments `a`,`x`, and `y` and a local `REAL` variable `h`. Suppose that we are interested in the first-order derivative information of $\mathtt{y} = (y_i)_{i=1,\ldots,10}$ with respect to $\mathtt{x} = (x_j)_{j=1,\ldots,10}$, i.e. the corresponding 10×10 Jacobian matrix (or simply Jacobian) $F' = (\partial y_i / \partial x_j)_{i,j=1,\ldots,10}$. The elements of `x` are called *independent variables* whereas the elements of `y` are said to be *dependent variables*. Both independent and dependent variables are set to be *active*. Furthermore, all variables whose value depends somehow on the value of an independent variable and which affect the value of some dependent variable are called active. Above, `h` is active while `a` remains *passive* since there is no dependence of its value on any of the independent variables. Knowing the set of all active variables at every single point in the program is crucial for the efficiency of AD. For large-scale applications applicability of AD is directly dependent on its efficiency. A static data flow analysis can be applied to achieve this [15].

Using the *forward mode* of AD [13, Chapter 3] a differentiated subroutine `D_SUB` is generated for `SUB` as follows.

```
do i=1,10
  if (i>1) then
    d_h=d_x(i)*y(i-1)+x(i)*d_y(i-1)
    h=x(i)*y(i-1)
  else
    d_h=a*d_x(i)+d_y(10)
    h=a*x(i)+y(10)
  endif
  d_y(i)=cos(h)*d_h
  y(i)=sin(h)
end do
```


SAC 2003 Melbourne, Florida USA

Derivative components `d_x, d_h, d_y` are associated with all active program variables and they are declared accordingly. Knowing how to compute the local partial derivatives of the *elementary* functions `*`, `-`, and `sin` for given values `a` and `x` one easily obtains `d_y` for given `d_x`. For example, we know that the partial derivative of `x(i)*y(i-1)` with respect to `x(i)` is `y(i-1)`. Similarly, differentiating `sin(h)` with respect to `h` yields `cos(h)`. The chain rule is used to combine these local partials in a mathematically correct way. In fact, forward mode AD computes `d_y` as the Jacobian times vector product $F' \cdot$ `d_x`. In order to accumulate the whole Jacobian column by column the values of `d_x` are chosen as the Cartesian basis vectors in $\mathbb{R}^{10}$. In other words, F' can be obtained at a computational cost which is proportional to the number of independent variables. Alternatively, *reverse mode* AD [13, Chapter 3] delivers the whole Jacobian row by row at a computational cost which is proportional to the number of dependent variables. This is of particular interest for the computation of large gradients (see *cheap gradient principle* in [13, Chapter 3]) where there is a single dependent variable only.

There are two approaches to implementing software tools for AD. *Operator overloading tools*, such as ADOL-C [3] and AD01 [1], exploit the operator overloading capabilities of languages such as C++ and Fortran 95. They define a new active data type containing the derivative component `d_h` in addition to the function value `h`. This corresponds to *doublets* (`h`, `d_h`) [13, Chapter 5]. All active variables must be redeclared correspondingly. The arithmetic operators and intrinsic functions are overloaded for arguments of the new active type, e.g. (`y(i), d_y(i)`) = (`sin(h), cos(h)*d_h`).

Source transformation tools, including ADIFOR [2], ADIC [4], and TAPENADE [5], implement compiler front-ends to generate a differentiated version of the original program as shown above. The code is analyzed both syntactically and semantically followed by a static activity analysis. AD is performed on the abstract internal representation, for example, by generating a differentiated version of the annotated syntax tree which is then unparsed. The result is a derivative code written in the same language as the original code. It must be integrated into existing applications via driver routines to be provided by the user. For example, the value of `d_x` must be initialized in addition to the original inputs in the example above.

Operator overloading tools for first and higher order derivatives are rather easy to implement and often very flexible and robust. On the downside, they use a derived data type that prevents the compiler from performing highly desirable code optimizations. The lack of a framework suitable for static data flow analysis forces the user of such tools to decide which program variables are active. The straight forward approach of making all floating-point variables active is often too much of an overestimation to be feasible. Selecting the right ones manually is hopeless for codes containing several thousands of program variables. For all these reasons, operator overloading tools for AD are often not well-suited for large-scale application programs.

In many cases the derivative code generated by source transformation tools for AD is more efficient. Activity analysis reduces the amount of trivial derivative computations considerably. The generated code is run through a compiler which can then produce optimized code. However, the development of such tools is considerably more difficult. Essentially, an industrial-strength compiler front-end is required whose implementation is everything but trivial. Robust and efficient static data flow analyses that minimize the overestimation resulting from their implicit conservatism must be provided [19]. Most available source transformation AD tools satisfy only a subset of these requirements.

2. FORTRAN 95 COMPILER AD

In this section we describe an approach which aims towards combining the advantages of both operator overloading and source transformation tools for AD. This work is the first stage of a collaborative research project between the Computer Science Department, University of Hertfordshire, Hatfield, and NAG Ltd., Oxford, UK. Funding has been provided by EPSRC (`www.epsrc.ac.uk`) under research grant number GR/R55252/01. The techniques described are implemented in an experimental version of the NAGWare Fortran 95 compiler which is planned to be released for beta-testing by the end of 2002. Examples and numerical results as well as further technical details can be found on the project's web site under

`www.nag.co.uk/nagware/research/ad_overview.asp`.

We consider compiler AD in Fortran 95 as a hybrid operator overloading / source transformation technique. The entire mathematical functionality is contained within a Fortran 95 module named `active_module.f95`. An opaque user-defined active data type `ACTIVE_TYPE` is provided together with overloaded versions for the elementary arithmetic operators and intrinsic functions. In the current version the user can select independent and dependent variables and compute the corresponding Jacobians. For example, the vector forward mode [13, Chapter 3] can be implemented using the following active type.

```
TYPE ACTIVE_TYPE
  REAL ::  v
  REAL, ALLOCATABLE, DIMENSION(:) :: d
END TYPE ACTIVE_TYPE
```

In addition to the function value a vector of derivative components can be allocated. In contrast to the scalar forward mode, this allows the computation of Jacobian times matrix products $F' \cdot S$ (instead of a Jacobian times vector product) by performing a single evaluation of the derivative code. To compute the whole Jacobian of a vector function $F : \mathbb{R}^n \to \mathbb{R}^m$ the `d` component of every independent scalar variable (or every scalar component of independent array variables) is initialized as a Cartesian basis vector such that S becomes equal to the identity matrix in $\mathbb{R}^n$. All elementary arithmetic operators and intrinsic functions are overloaded to handle the new data type in a mathematically correct way. For example, a sine of an active scalar variable is implemented as follows.

```
ELEMENTAL FUNCTION sin_ACTIVE(arg1)  RESULT(res)
  TYPE(ACTIVE_TYPE), INTENT(IN) :: arg1
  TYPE(ACTIVE_TYPE)             :: res

  res%v = sin(arg1%v)
  IF ( ALLOCATED( arg1%d ) ) THEN
    ALLOCATE( res%d(1:UBOUND(arg1%d,1)) )
```

```
    res%d = arg1%d * cos( arg1%v )
  END IF

END FUNCTION sin_ACTIVE
```

The function value `res%v` is computed as usual. Additionally, the values of the derivative components are evaluated using the partial derivative of sine with respect to its argument and the chain rule. Excerpts from `active_module.f95` can be found on our web site.

When using the differentiation-enabled NAGWare Fortran 95 compiler the example from the previous section is modified as follows.

```
TYPE(DERIV_TYPE) :: jac
...
DIFFERENTIATE
INDEPENDENT(x)
do i=1,10
  if (i>1) then
    h=x(i)*y(i-1)
  else
    h=a*x(i)+y(10)
  endif
  y(i)=sin(h)
end do
jac=JACOBIAN(y,x)
END DIFFERENTIATE
write(*,*) DERIVTOREAL(jac)
```

With the `-ad` compiler switch set differentiation takes place inside the active section which is framed by `DIFFERENTIATE ... END DIFFERENTIATE`. The variable `x` is set to be independent and the Jacobian of `y` with respect to `x` is obtained by calling `JACOBIAN(y,x)`. The result of `JACOBIAN` is of type `DERIV_TYPE`. The conversion function `DERIVTOREAL` extracts the `REAL` values which we simply print in this example. We aim to keep the user interface as simple as possible. This is how the differentiation enabled version of the NAGWare Fortran 95 compiler handles this code in principle: Lexical, syntax, and semantic analysis results in some form of an annotated syntax tree and a symbol table; `x` is marked as independent; `y` is marked as dependent; a, so far, very simple static activity analysis marks `h` as active and recognizes that `a` remains passive; active copies of type `ACTIVE_TYPE` are generated for all active variables; their values are initialized as the original values; the derivative component of the `x(i)` are initialized as the Cartesian basis vectors $e_{\mathtt{i}} \in I\!R^{10}$ for $\mathtt{i} = 1, \ldots, 10$; inside the active section the syntax tree is modified such that all computations are performed on the active copies using the overloaded operators and intrinsic function defined in `active_module.f95`. The derivative access function `JACOBIAN` is also defined in `active_module.f95` which can be viewed in part on our web site. Internally, the compiler has been extended to accept the essential new language constructs. Both `ACTIVE_TYPE` and `DERIV_TYPE` are "known" to the compiler and we have access to the operator overloading resolution phase. In principle, this enables us to generated efficient derivative code based on activity analysis and intermediate code optimizations. The latter, are the subject of ongoing work toward a second prototype to be released by summer 2003. It will feature both statement- [8, 16] and basic-block-level preaccumulation [17, 20] of local gradients and Jacobians. So far, our main focus has been on establishing the interface between the compiler and AD algorithms. Potentially, code optimization algorithms can be developed not only at the intermediate code level but even for specific architectures. In general, this is based on the fact that derivative code is like any other code. Whatever is good for code in general must be good for derivative code in particular. However, for the latter we have additional information available which can and should be exploited for the development of specialized code optimization algorithms.

As pointed out before, the efficiency of derivative code depends strongly on the quality of the activity analysis. Static analysis often results in a considerable overestimation of the active set. Combining it with dynamic analysis can overcome this problem at the cost of an overhead that depends on the quality of the static analysis itself. This can be achieved by using the sparse version of `ACTIVE_TYPE` as described below.

A further improvement has been achieved from the point of view of adding new forward mode AD functionality to the compiler, for example the capability to compute higher order derivatives using truncated Taylor series [9] or (value, gradient, Hessian matrix) triplets [14]. All we have to do is write a new `active_module.f95` which defines a corresponding active data type and implements the overloaded operations, intrinsic functions and derivative access functions.

Our approach allows us to exploit sparsity in the computation at various levels. Seed matrix compression techniques [12, 18] can be applied in vector forward mode if the Jacobian itself is sparse. An example is discussed in the appendix. In sparse forward mode [13, Chapter 6] the `d` components are implemented as sparse data structures (for example, as (index, value) pairs) which allows to exploit sparsity of the problem at an elemental level. Furthermore, the result of activity analysis can be used to exploit a likely symbol-level sparsity. Therefore, we implemented another active data type as follows.

```
TYPE GRAD_TYPE
  REAL, DIMENSION(:), ALLOCATABLE :: g
END TYPE GRAD_TYPE

TYPE ACTIVE_TYPE
  REAL :: v
  TYPE(GRAD_TYPE), DIMENSION(:), ALLOCATABLE :: d
END TYPE ACTIVE_TYPE
```

Every active variable contains a `d` vector with a number of components of type `GRAD_TYPE` that is equal to the number of independent program variables. In our example this number is equal to one since `x` is the only independent program variable. Symbol-level sparsity is not an issue in this simple case. In general, the single components of `d` are allocated as `g` vectors with a size that equals the number of scalar elements in the corresponding independent program variable. If activity analysis tells us that an active variable does not depend on some independent variable then the corresponding `d` entry is simply not allocated. This decision can either be made at compile time which corresponds to the classical static activity analysis as described in [15] or it can be made dynamically by including a check on allocated `d` components into the definition of the overloaded operators and intrinsic

functions. In a dynamic activity analysis framework the d component of a passive variable that could not be identified as passive by the static analysis remains deallocated.

3. CONCLUSION AND OUTLOOK

We believe that in order to generate highly robust and efficient derivative code the corresponding AD algorithms have to be incorporated into existing industrial strength compilers. Potentially, this approach gives us access to all stages of the compilation process allowing us to perform AD specific static analyses and optimizations of the derivative code. Furthermore, hybrid static and dynamic data flow analyses can be implemented.

So far, we have been concentrating on incorporating forward mode AD algorithms into the NAGWare Fortran 95 compiler. We consider this as the first step towards a fully functional adjoint compiler for differentiating numerical programs using the reverse mode of AD. Although the basic infrastructure has been established a considerable effort will be necessary to achieve this goal.

4. ACKNOWLEDGEMENTS

Naumann was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-ENG-38.

Riehme was supported by the Engineering and Physical Sciences Research Council, U.K., under grant GR/R55252/01.

APPENDIX

A. BRATU PROBLEM WITH SEEDING

We consider a variant of the Bratu problem from the MINPACK test problem collection [6] given as the following Fortran routine.

```
SUBROUTINE EXPL(dim,parmax,x,prm,F)
  INTEGER, INTENT(IN) :: dim, parmax
  REAL, INTENT(IN)    :: x(dim), prm(parmax)
  REAL, INTENT(INOUT) :: F(dim)
  INTEGER             :: i
  REAL                :: h

  h = 2.0/(dim+1)
  F(1) = -2*x(1)+h*h*prm(1)/12.0* &
  & (1+10*exp(x(1)/(1.0+prm(2)*x(1))))
  F(2) = x(1)+h*h*prm(1)/12.0* &
  & exp(x(1)/(1.0+prm(2)*x(1)))

  DO i=2,dim-1
     F(i-1) = F(i-1)+x(i)+h*h*prm(1)/ &
     & 12.0*exp(x(i)/(1.0+prm(2)*x(i)))
     F(i) = F(i)-2*x(i)+h*h*prm(1)/ &
     & 1.2*exp(x(i)/(1.0+prm(2)*x(i)))
     F(i+1) = x(i)+h*h*prm(1)/12.0* &
     & exp(x(i)/(1.0+prm(2)*x(i)))
  END DO

  F(dim-1) = F(dim-1)+x(dim)+h*h*prm(1)/ &
  & 12.0*exp(x(dim)/(1.0+prm(2)*x(dim)))
  F(dim) = F(dim)-2*x(dim)
  F(dim) = F(dim)+h*h*prm(1)/12.0* &
     & (1+10*exp(x(dim)/(1.0+prm(2)*x(dim))))
  END SUBROUTINE EXPL
```

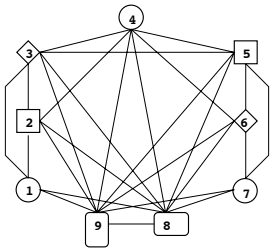
Our objective is to compute the Jacobian matrix F' of F with respect to both x and prm using Curtis-Powell-Reid (CPR) seeding [12]. In our example we set dim $= 7$ and parmax $= 2$. Using the vector forward mode provided by the NAGWare Fortran 95 compiler we get the following 7×9 Jacobian:

```
-1.89  1.01  0.00  0.00  0.00  0.00  0.00  0.21 -0.48
 1.01 -1.87  1.01  0.00  0.00  0.00  0.00  0.40 -1.78
 0.00  1.01 -1.87  1.01  0.00  0.00  0.00  0.49 -2.70
 0.00  0.00  1.01 -1.87  1.01  0.00  0.00  0.56 -3.50
 0.00  0.00  0.00  1.01 -1.87  1.01  0.00  0.49 -2.70
 0.00  0.00  0.00  0.00  1.01 -1.87  1.01  0.40 -1.78
 0.00  0.00  0.00  0.00  0.00  1.01 -1.89  0.21 -0.48
```

It is obtained by initializing the joint derivative components of the independent variables x and prm as a 9×9 identity matrix. Notice, that F' is sparse. In fact, a compressed version can be computed by *seeding* the derivative components of the independent variables as a 9×5 matrix S with rows representing Cartesian basis vectors in $\mathbb{R}^5$.

CPR seeding is based on the idea that certain columns of the Jacobian can be merged to share storage. For example, column 1 and column 7 could share one column, thus resulting in a compressed version of the Jacobian. This implies that the sparsity pattern must be known in advance in order to exploit matrix compression techniques. Remember that the derivative code generated by the compiler in dense vector forward mode always loops over the derivative components of all active variables. Many of their entries are equal to zero, leading to predictably trivial multiplications that one would like to avoid. Therefore, instead of computing the Jacobian as a Jacobian times identity matrix product, one could try to compute a compressed Jacobian using a seed matrix with fewer columns than the identity. Since the number of independent variables is often very large, the size of the seed matrix can become much smaller, leading to a decreased complexity of the Jacobian accumulation.

In CPR seeding, one considers the column incidence graph of the Jacobian to try to determine a minimal vertex coloring. Whenever two vertices share the same color, the corresponding columns can be stored in the same column of the compressed Jacobian. In our example, the column incidence graph has the following structure.



Unfortunately, since the vertex coloring problem is known to be NP-complete in general, the use of heuristics is essential. The coloring in the example graph has been found "by inspection". Different colors are represented by different vertex shapes. The number ν of different colors used determines the number of columns in the CPR seed matrix. Its rows are Cartesian basis vectors in $\mathbb{R}^\nu$. In our case $\nu = 5$. Whenever two vertices share the same color, the corresponding rows in the seed matrix contain the same Cartesian basis vector. The compressed Jacobian B can be computed using

the new features of the NAGWare Fortran 95 compiler as shown below.

```
PROGRAM MAIN

  IMPLICIT NONE

  INTEGER, PARAMETER :: dim = 7, parmax = 2
  INTEGER, PARAMETER :: n = dim+parmax
  INTEGER            :: i
  REAL               :: x(dim), prm(parmax)
  REAL               :: f(dim)
  TYPE(DERIV_TYPE)   :: jac
  REAL               :: seed1(9,5)

  x   = (/ 1.72, 3.45, 4.16, 4.87, 4.16, &
         & 3.45, 1.72 /)
  prm = (/ 1.3, 0.245828  /)

  seed1 = 0
  seed1(1,1)  = 1; seed1(2,2)  = 1; seed1(3,3)  = 1
  seed1(4,1)  = 1; seed1(5,2)  = 1; seed1(6,3)  = 1
  seed1(7,1)  = 1; seed1(8,4)  = 1; seed1(9,5)  = 1

  DIFFERENTIATE
  INDEPENDENT( x, SEED=seed1(1:7,:) )
  INDEPENDENT( prm, SEED=seed1(8:9,:) )

  call EXPL( dim, parmax, x , prm, f )

  jac = JACOBIAN( f )
  END DIFFERENTIATE

  CALL WRITE_JACOBIAN( jac )

CONTAINS

  SUBROUTINE EXPL(dim,parmax,x,prm,F) ...

END PROGRAM MAIN
```

Both the inputs and the seed matrix are initialized. The latter is then divided into the two parts corresponding to `x` and `prm` and passed as arguments of the respective `INDEPENDENT` statements. Everything else is take care of internally.

Compiling the above with the `-ad` compiler switch set results in the generation of code that produces the following output.

```
-1.89  1.01  0.00  0.21 -0.48
 1.01 -1.87  1.01  0.40 -1.78
 1.01  1.01 -1.87  0.49 -2.70
-1.87  1.01  1.01  0.56 -3.50
 1.01 -1.87  1.01  0.49 -2.70
 1.01  1.01 -1.87  0.40 -1.78
-1.89  0.00  1.01  0.21 -0.48
```

F' itself can be restored by a simple back-substitution process as explained, for example, in [13, Chapter 7].

B. REFERENCES

[1] `www.cse.circ.ac.uk/Activity/HSL`. url.

[2] `www.cs.rice.edu/~adifor`. url.

[3] `www.math.tu-dresden.de/wir/project/adolc`. url.

[4] `www.mcs.anl.gov/~adic`. url.

[5] `www-sop.inria.fr/tropics`. url.

[6] B. Averik, R. Carter, and J. Moré. The MINPACK-2 test problem collection (preliminary version). Technical Report 150, MCS, ANL, 1991.

[7] M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools*, Proceedings Series, Philadelphia, 1996. SIAM.

[8] C. Bischof, A. Carle, P. Khademi, and A. Maurer. The ADIFOR 2.0 system for Automatic Differentiation of Fortran 77 programs. *IEEE Comp. Sci. & Eng.*, 3(3):18–32, 1996.

[9] G. Corliss. Overloading point and interval Taylor operators. In *[11]*, pages 139–146. SIAM, 1991.

[10] G. Corliss, C. Faure, A. Griewank, L. Hascoet, and U. Naumann, editors. *Automatic Differentiation of Algorithms – From Simulation to Optimization*, New York, 2002. Springer.

[11] G. Corliss and A. Griewank, editors. *Automatic Differentiation: Theory, Implementation, and Application*, Proceedings Series, Philadelphia, 1991. SIAM.

[12] A. Curtis, M. Powell, and J. Reid. On the estimation of sparse Jacobian matrices. *J. Inst. Math. Appl.*, 1974.

[13] A. Griewank. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation.* Number 19 in Frontiers in Applied Mathematics. SIAM, Philadelphia, 2000.

[14] R. Hammer, M. Hocks, U. Kulisch, and D. Ratz. *Numerical Toolbox for Verified Computing I – Basic Numerical Problems.* Springer, Heidelberg, 1993.

[15] L. Hascoët, U. Naumann, and V. Pascual. TBR analysis in reverse-mode Automatic Differentiation. *Elsevier Science (under review)*, 2002.

[16] U. Naumann. On optimal jacobian accumulation for single expression use programs. Preprint ANL/MCS-P944-0402, Argonne National Laboratory, April 2002. Under review for Mathematical Programming.

[17] U. Naumann. Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. Preprint ANL/MCS-P943-0402, Argonne National Laboratory, April 2002. Excepted for publication in Mathematical Programming.

[18] G. Newsam and J. Ramsdell. Estimation of sparse Jacobian matrices. *SIAM J. Alg. Dis. Meth.*, 4:404–417, 1983.

[19] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation.* ACM, 2000.

[20] M. Tadjouddine, S. Forth, and J. Pryce. AD tools and prospects for optimal AD in CFD flux calculations. In *[10]*.